Efficient vectors with static and dynamic storage in C++

Hubert Obrzut

Institute of Compute Science, University of Wrocław

Thursday 17th February, 2022

Presentation plan

1. *vector* overview
2. Static storage variant
3. Dynamic storage variant
4. Overriding type triviality
5. Summary

vector overview

Characteristics

- simple purpose: storage for objects, managing their lifetime and random access
- the most commonly used data structure in C++
- several variants and implementations

vector overview

Characteristics

- simple purpose: storage for objects, managing their lifetime and random access
- the most commonly used data structure in C++
- several variants and implementations

Goal: efficient implementation of two variants: with static and dynamic storage.

static_vector data structure

```
template<class T, size_t C>
class static_vector<T, C>;
```

Listing 1: `static_vector declaration`

Overview

- static storage
- dynamically-resizeable

Usage

Useful when:

- number of elements to store is known beforehand
- dynamic allocation is not acceptable, e.g. performance reasons
- objects cannot be default constructed (example below)

Usage example

```
T array[N];
for (int i = 0; i < N; ++i)
    array[i] = T(custom, arguments);

static_vector<T, N> v;
for (int i = 0; i < N; ++i)
    v.emplace_back(custom, arguments);
```

In C-style array case, we unnecessary create default constructed objects.

Existing implementations

`static_vector` implementations

- Boost
- Folly
- EASTL
- `uwr::static_vector`

Existing implementations

`static_vector` implementations

- Boost
- Folly
- EASTL
- `uwr::static_vector`

Note

No STL implementation.

Alternative implementations

```
template<class T, size_t C>
class static_vector {
    ...
    size_type m_size;
    ...
};
```

Listing 2: *size-based* variant.

```
template<class T, size_t C>
class static_vector {
    ...
    T* m_end;
    ...
};
```

Listing 3: *pointer-based* variant.

Alternative implementations cont.

size-based variant advantages

- slightly faster in the average case
- more natural
- minimal size dispatch
- *trivially-relocatable*

```
// size-based variant
assert(sizeof(static_vector<char, 5>) == 6);
```

Listing 4: Occupies 6 bytes instead of 16.

uwr::static_vector advantages

Improvements

- implemented as a separate type – optimized for static storage
- cache-friendly
- push_back function optimization
- economical size

Additional features

- C++20 compliant, C++17 compatible – operator<=>, std::erase, std::erase_if
- constexpr support
- fast_push_back, fast_emplace_back, safe_pop_back

Benchmark

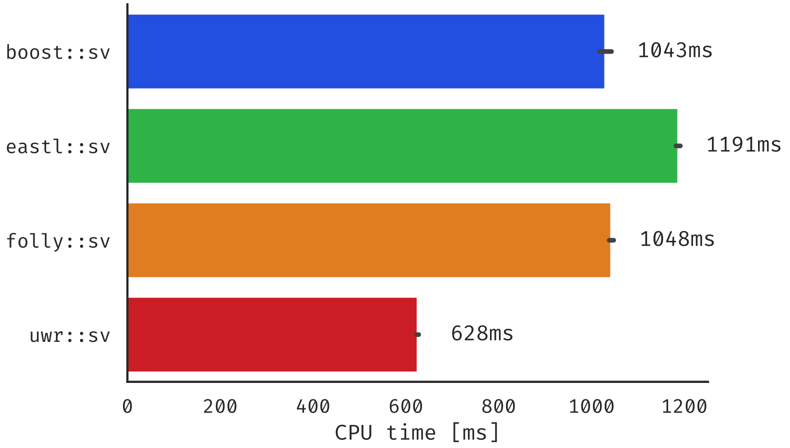
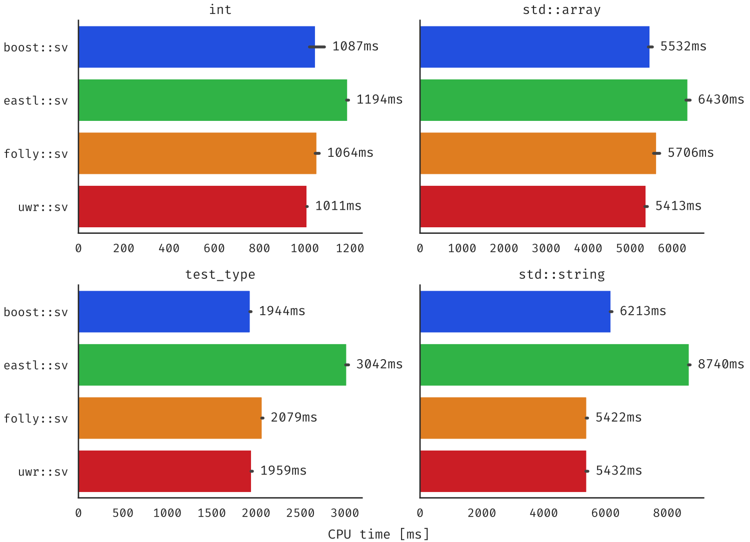


Figure: Benchmark example comparing `uwr::static_vector` with alternative implementations.

Benchmark


int
boost::sv
eastl::sv
folly::sv
uwr::sv
1087ms
1194ms
1064ms
1011ms
0 200 400 600 800 1000 1200
std::array
5532ms
6430ms
5706ms
5413ms
0 1000 2000 3000 4000 5000 6000
test_type
1944ms
3042ms
2079ms
1959ms
0 500 1000 1500 2000 2500 3000
std::string
6213ms
8740ms
5422ms
5432ms
0 2000 4000 6000 8000
CPU time [ms]

Real code example

Chess benchmark

	Number of playouts	Number of states
`boost::static_vector`	625	195k
`uwr::static_vector`	765	240k

Aprox. 20% speedup.

vector data structure

Overview

- most commonly used variant of *vector*
- dynamically-resizeable with dynamic storage
- keeps pointer to the allocated memory block
- continuous storage
- usually allocates more memory than needed
- uses growth policy for reallocation

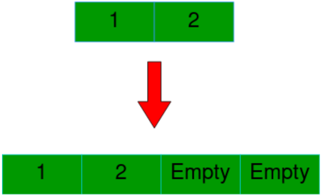


Figure: *vector*'s growth example

Existing implementations

`vector` implementations

- STL
- Boost
- EASTL
- Folly
- `rvector`

uwr::vector implementation

Characteristics

- improves cache-friendliness
- uses memory mappings to allocate memory
- mmap, mremap, munmap

mremap system call

- can expand mapping in-place
- very efficient for *trivial* types

```
return (T*)mremap(data,
                  old_capacity * sizeof(T),
                  new_capacity * sizeof(T),
                  MREMAP_MAYMOVE);
```

Listing 5: *trivial* type reallocation using mremap

Reallocation strategy for *non-trivial* types

```
return (T*)mremap(data,
                  old_capacity * sizeof(T),
                  new_capacity * sizeof(T),
                  0);
```

Listing 6: *non-trivial* type reallocation using `mremap`

Strategy

- `mremap` fails to expand in-place
- binary search the maximal sucessful growth over the number of pages

Reallocation strategy for *non-trivial* types

```
return (T*)mremap(data,
                  old_capacity * sizeof(T),
                  new_capacity * sizeof(T),
                  0);
```

Listing 7: *non-trivial* type reallocation using `mremap`

Strategy

- `mremap` fails to expand in-place
- binary search the maximal sucessful growth over the number of pages

Results

Increased number of sucessful reallocations, e.g., from 24% to 45%.

Benchmark

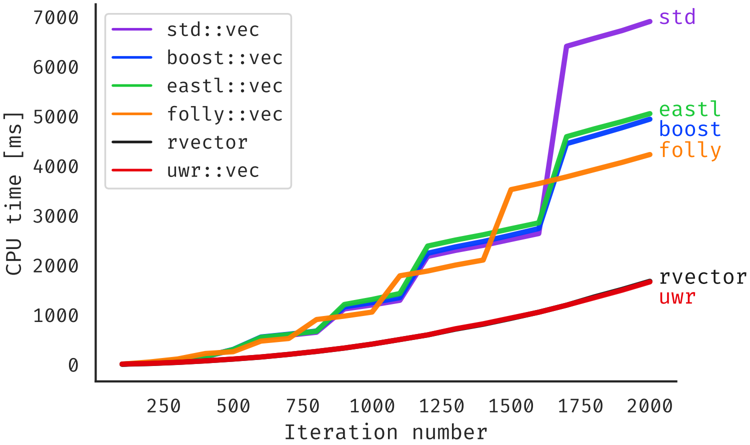


Figure: Example benchmark comparing `uwr::vector` with alternative implementations for *trivial* type.

Benchmark

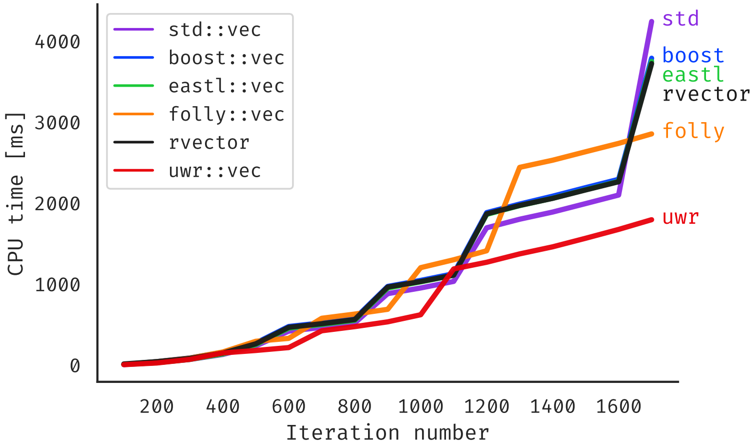


Figure: Example benchmark comparing `uwr::vector` with alternative implementations for *non-trivial* type.

Benchmark

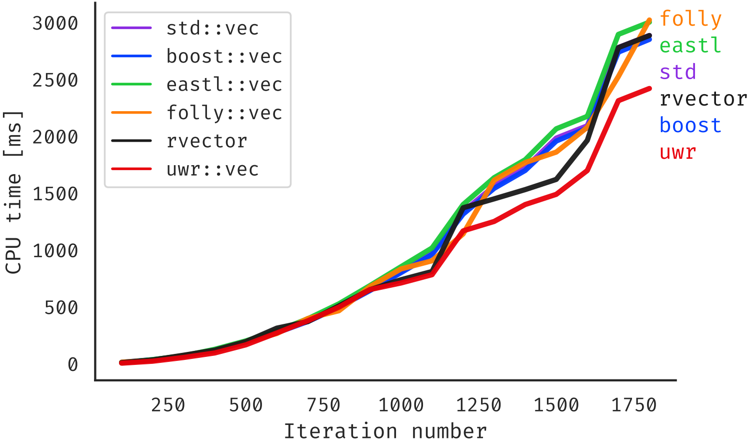


Figure: Example benchmark comparing `uwr::vector` with alternative implementations for *non-trivial* type with larger number of vectors in the environment.

Custom growth rate

```
uwr::vector<T, std::ratio<7, 5>> v;
```

Listing 8: Using uwr::vector with a custom growth rate.

Type triviallity

trivially-relocatable type

Type that holds all of its invariants when moved to another memory location with raw copy.

Note

Still can have defined `move` constructor/operator.

Examples

trivial types

- `uwr::vector`, `std::vector`, ...

Examples

trivial types

- `uwr::vector`, `std::vector`, ...
- `std::unordered_map`
- `std::unordered_set`

Examples

trivial types

- `uwr::vector`, `std::vector`, ...
- `std::unordered_map`
- `std::unordered_set`
- `std::unique_ptr`, `std::shared_ptr`

Examples

trivial types

- `uwr::vector`, `std::vector`, ...
- `std::unordered_map`
- `std::unordered_set`
- `std::unique_ptr`, `std::shared_ptr`
- big `std::string` (without SSO)

Examples

trivial types

- `uwr::vector`, `std::vector`, ...
- `std::unordered_map`
- `std::unordered_set`
- `std::unique_ptr`, `std::shared_ptr`
- big `std::string` (without SSO)

Note

`std::map` and `std::set` are not.

Overriding type triviality

```
namespace uwr::mem {

template<>
inline constexpr bool is_trivially_relocatable_v<
    MySpecialType> = true;

}
```

Listing 9: Overriding type triviality for `uwr::vector`

Overriding type triviality

```
namespace uwr::mem {

template<>
inline constexpr bool is_trivially_relocatable_v<
    MySpecialType> = true;

}
```

Listing 10: Overriding type triviality for `uwr::vector`

Note

`folly::vector` supports this feature as well.

Benchmark

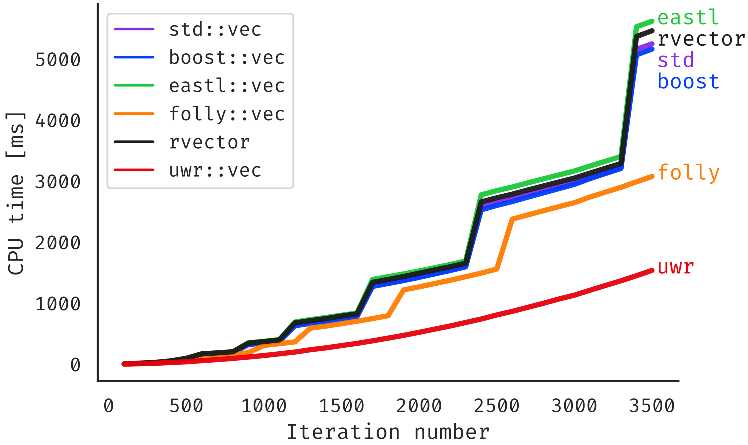


Figure: Example benchmark comparing `uwr::vector` with alternative implementations for type with overriden triviallity.

Summary

Both vectors

- C++20 compliant, C++17 compatible
- cache-friendly
- no external dependencies

`uwr::static_vector`

- economical size
- has `push_back` optimization

`uwr::vector`

- allows overriding type triviality
- has custom *growth factor* support
- efficient for *trivial* and *non-trivial* types

Thank you for your attention

Both vectors

- C++20 compliant, C++17 compatible
- cache-friendly
- no external dependencies

`uwr::static_vector`

- economical size
- has `push_back` optimization

`uwr::vector`

- allows overriding type triviality
- has custom *growth factor* support
- efficient for *trivial* and *non-trivial* types